Fulfilling the *Need for Speed*: A brief introduction to parallel processing in the R environment.

As published in Benchmarks RSS Matters, March 2014

http://web3.unt.edu/benchmarks/issues/2014/03/rss-matters

Jon Starkweather, PhD

Jon Starkweather, PhD
jonathan.starkweather@unt.edu
Consultant
Research and **S**tatistical **S**upport

http://www.unt.edu

http://www.unt.edu/rss

RSS hosts a number of “Short Courses”.
A list of them is available at:
http://www.unt.edu/rss/Instructional.htm

Those interested in learning more about R, or how to use it, can find information here:
http://www.unt.edu/rss/class/Jon/R_SC

Fulfilling the *Need for Speed*: A brief introduction to parallel processing in the R environment.

This month we provide a brief overview of some methods available for applying parallel processing to the R environment. There are essentially two ways of parallel processing when using R. The first, and most frequently used, involves utilizing the multiple cores (or processors) within a single computer. Given the widespread availability of multicore computers on campuses and in research labs now, it is common to have access to such a machine. There are, and have been for a while, R packages which allow the user to utilize more than core on such a machine (only one core is used by a default installation of R). The second method for parallel processing in the R environment involves the use of distributed computing, such as multiple computers of a network or a High Performance Computer (HPC) - which has multiple nodes or modules controlled by a central operating system.

The current research culture is saturated with the phrase *Big Data*; it seems everyone is obsessed with the size of their data, or the analysis of large data. The current research climate also appears to be recognizing the complexity which seems ubiquitous in nature — harkening back to ideas long dormant; such as complexity theory (Weaver, 1948; Waldrop, 1992), general systems theory (Bertalanffy, 1968), and cybernetics (Wiener, 1965) — meaning it is no longer acceptable to perform a simple textbook style analysis (e.g. multiple regression analysis, ANOVA, etc.). Often journals are expecting complex, multidirectional and multidimensional models which account for the complex interrelationships among variables of interest (moderation, mediation, SEM, HLM, neural networks, etc.) as well as computer intensive optimization methods for estimating the parameters of these complex models (e.g. MCMC, estimation maximization / maximum likelihood optimization, genetic optimization, ant colony optimization, etc.). It is also well known that computing technology (i.e. processing speeds, memory and storage device sizes, & data transfer rates) have increased at a rate consistent with a Power Law, sometimes called Moore's Law (Moore, 1965). Therefore, it is no surprise the need for greater computing resources has led to the development of software which allows researchers to fit complex models to large data sets while optimizing parameter estimates using all available computing resources - whether those resources are included in one machine or distributed across multiple machines in a network or HPC setting.

As is the case with most things, the statistical programming language environment R offers a choice among several packages - each of which provides a method for *parallelizing* one's data analysis. Support for parallel processing in the R environment began with R 2.14.0 (Eddelbuettel, 2013) with the inclusion of a (then) new 'parallel' package (Ripley, Tierney and Urbanek, 2014) in all base installations of R (Hornik, 2014). Below we offer a simple introduction to parallel processing using multiple cores or processors on a single machine. We then provide some resources for distributed parallel processing (e.g. in an HPC or network setting).

1 Single Core (default) Processing

In order to demonstrate the benefits of parallel processing, we need a way of measuring the speed with which R completes a discrete task. Below, we will be using the 'system.time' function to measure the number of seconds between initiation and completion of a submitted function or set of functions. The 'system.time' function works by monitoring the computer's internal system clock and returning the difference (in seconds) between the initiation and completion of a process. Below, we use the 'system.time' function as a wrapper around any task we are attempting to measure (the speed of that embedded task)

and request the function to return the "elapsed" time (in seconds) which is the 3rd element of the output from that function.

For the purposes of demonstration and simulation; let's consider an imaginary lock which requires a three integer combination. We can make the lock easy or difficult to *pick* by increasing or decreasing the number of possible choices for each integer of the combination. For example, a *brute force* search algorithm will quickly find the correct combination if there are only 5 choices (1, 2, 3, 4, 5 with replacement) for each of the three integers of the combination. However, if there are 50 choices (1, 2, 3, ..., 49, 50 with replacement) for each of the three integers, then it will require significantly more time (and more computing resources) to solve.

So, we first need to create a combination (abbreviated as "combin" below) for our imaginary lock. In order to do this, we select the number of choices (abbreviated as "choi" below) for each of the three integers of our combination (with replacement). Then, we sample 3 integers with replacement from a sequence of values from 1 to 50 (choices) to create our combination.

```
choi <- 50; choi
[1] 50
combin <- sample(seq(1:choi), 3, replace = T); combin
[1] 21  6  19
```

We can see in the example above, our combination (combin) is 21, 6, 19; each selected from a sequence of 1 to 50.

Next, we create a function (or operation) which we will use to simulate a computer resource intensive process. Our testing function is nothing more than a *brute force* search to discover the preset combination of an imaginary lock — thus the function's name 'lockpick.fun'. Brute force refers to the exhaustive, sequential, search of every possible solution.

```
lockpick.fun <- function(combination, choices){
  ch <- seq(1:choices)
  solution.matrix <- expand.grid(ch, ch, ch)
  i <- 0
  picked <- "FALSE"
  while(picked == "FALSE"){
      i <- i + 1
      draw <- solution.matrix[i,]
      if(combination[1] == draw[1] & combination[2] == draw[2] &
         combination[3] == draw[3]){
         picked <- "TRUE"; print("PICKED!!")}
      }
  out <- paste("Number of iterations =", i, sep = " ")
  return(out)
  }
```

Notice above the function requires two arguments. First, the user supplied vector containing the three integer combination to the imaginary lock ("combination"). Second, the user must supply the integer value representing the number of choices available for each integer in the three integer combination ("choices"). We set the number of choices earlier when we created the combination (50; "choi"). As stated above, we can increase this number of choices (or decrease it) when creating the combination in order to make the brute force search operation more (or less) computer resource intensive.

Next, we show an example of the 'system.time' function when used with an application of the above specified lock picking function — given the combination above based on 50 choices for each of the 3

combination integers.

```
system.time(test.1 <- lockpick.fun(combination = combin, choices = choi))
[1] "PICKED!!"
   user  system elapsed
  28.27    2.61   31.24
test.1
[1] "Number of iterations = 45271"
```

So, we see above our Window's 7 (64-bit) desktop computer running R 3.0.2 (64-bit) used 31.24 seconds to complete the lock pick function when each of 3 integers were chosen from a sequence of 1 to 50 possible values. Keep in mind, the more resource intensive the task, the greater the benefit from parallel processing. For instance, the widely used for-loop is method of evaluating a function (or multiple functions) iteratively. Therefore, in order to make our example more demonstrative, we will require three runs, or iterations, of our lock picking function. To establish a *baseline* (in terms of speed), we will use the ubiquitous for-loop structure. Again, the default installation of R uses a single core or processor (even on machines with multiple processors).

Below you can see we are using the same combination (21, 6, 19) and number of choices (50) as was done above, but we are using a for-loop to repeat (or iterate) the solution 3 times. We are doing this so that the reader will see a more substantial decrease in processing time (than if we only applied the lock pick function once — the example function is rather quick and therefore attempting to parallel process it offers virtually no benefit).

```
b.results <- as.list(0)
b.time <- system.time(for (i in 1:3){
  b.results[[i]] <- lockpick.fun(combination = combin, choices = choi)
  })[3]
[1] "PICKED!!"
[1] "PICKED!!"
[1] "PICKED!!"
b.results
[[1]]
[1] "Number of iterations = 124235"

[[2]]
[1] "Number of iterations = 124235"

[[3]]
[1] "Number of iterations = 124235"

b.time
elapsed
 253.33
```

So, the above output indicates it took our single core (default R installation) 253 seconds to complete 3 iterations of our lock picking function. The "b.time" is simply an object containing the baseline amount of time required.

2 Multicore Processing

The vast majority of data analysis needs, regardless of data size or analysis complexity, can be satisfied using a single desktop computer with multiple cores. Substantial decreases in processing time can be observed when comparing the default single core operation to the same operation when utilizing two or more cores of the same machine. Again, the larger the job, the greater the benefit will be when using multiple cores.

Below we are going to be using the 'registerDoParallel' function of the 'doParallel' package (Weston, 2014) to register or recognize our machine's multiple cores (this machine has two cores). The 'doParallel' package has a three dependent packages ('foreach', 'iterators', & 'parallel'), one of which ('foreach')we will be using later to parallelize our 3 iterations of the lock pick function across the multiple cores.

```
library(doParallel)
Loading required package: foreach
foreach: simple, scalable parallel programming from Revolution Analytics
Use Revolution R for scalability, fault tolerance and more.
http://www.revolutionanalytics.com
Loading required package: iterators
Loading required package: parallel
```

Next, we register the two cores of our current machine (you could of course, register more cores if your machine has more than two.

```
registerDoParallel(cores = 2)
```

Next, we can run 3 iterations of our lock pick function using the 'foreach' function of the 'foreach' package (Weston, 2013). The 'foreach' function below looks remarkably like the for loop from above. First, the number of iterations (i) are specified (1:3), then the 'dopar' (do parallel) operator is used to specify what is supposed to be iterated.

```
t.time <- system.time(t.results <- foreach(i = 1:3) %dopar%
b.time <- system.time(for (i in 1:3){
  lockpick.fun(combination = combin, choices = choi))[3]
t.results
[[1]]
[1] "Number of iterations = 124235"

[[2]]
[1] "Number of iterations = 124235"

[[3]]
[1] "Number of iterations = 124235"

t.time
elapsed
 200.91
```

So we can see our multicore processing — using two cores — required only 201 seconds to complete the 3 iterations of our lock picking function. In order to compare this with the baseline time, we can perform a simple percentage transformation using the 'b.time' (baseline time) and 't.time' (test time) objects:

```
(b.time - t.time)/b.time
  elapsed
```

```
0.2069238
```

We can report a 21% decrease in processing time when using the 'foreach' multicore processing method compared to the baseline single core processing method. The example above may seem to provide a rather paltry improvement; however, consider a task which takes 12 hours with a single core and may only take 2.4 hours to complete with two cores — and that's assuming the same 20% increase in speed. Again, recall the larger the job (i.e. the more intensive the task), the greater the benefit of using multiple cores.

3 Distributed Computing

It is not feasible to provide an example of parallel processing with an HPC within the space limitations for this document. However, those interested in utilizing the UNT HPC (called Talon or Talon 2.0) are encouraged to visit the HPC web site[1] in order to set up an account (required for use of HPC and related resources). There are a variety of R packages designed to facilitate use of R in an HPC environment. In fact, there is an entire CRAN Task View devoted to high performance computing (Eddelbuettel, 2013). The HPC Task View provides descriptions of packages and their functions which are related to high performance computing (e.g. package snow: Simple Network of Workstations; "provides an abstraction layer by hiding the communications details."). Readers are strongly encouraged to review the Task View prior to contacting UNT's HPC service personnel.

4 Conclusions

Obviously, the main idea of this article it to make researchers aware of the multiple tools available to working with very large data, very complex models, and other computer resource intensive analysis. The R community has risen to the challenge of *big data* and continues to do so. Most data analysis needs can be satisfied using a single desktop computer with multiple cores (or processors) - which are readily available on campus. A simple example was provided to show the reader how easy it is to utilize multiple cores to speed up analysis. For more information on what R can do, please visit the Research and Statistical Support Do-It-Yourself Introduction to R[2] course website. An Adobe.pdf version of this article can be found here[3].

Until next time; *I'm sorry Dave. I'm afraid I can't do that.*

References & Resources

Bertalanffy, L. (1968). *General system theory: Foundations, development, applications*. New York: George Braziller Inc.

[1] `http://hpc.unt.edu/`
[2] `http://www.unt.edu/rss/class/Jon/R_SC/`
[3] `http://www.unt.edu/rss/rssmattersindex.htm`

Eddelbuettel, D. (2013). CRAN Task View: High-Performance and Parallel Computing with R. Available at:
`http://cran.r-project.org/web/views/HighPerformanceComputing.html`

Hornik, K. (2014). R FAQ (section 5.1.1 R Add-On Packages): List of packages which are included with an R distribution. Available at: `http://cran.r-project.org/faqs.html`

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics Magazine*, p 4.

Ripley, B., Tierney, L., & Urbanek, S. (2014). Package parallel. Manual available at:
`http://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf`

Waldrop, M. M. (1992). *Complexity: The emerging science at the edge of order and chaos*. New York: Touchstone (a division of Simon & Schuster Inc.).

Weaver, W. (1948). Science and Complexity. *American Scientist, 36*(4), 536-44.

Weston, S., [Revolution Analytics]. (2014). Package doParallel. Documentation available at CRAN:
`http://cran.r-project.org/web/packages/doParallel/index.html`

Weston, S., [Revolution Analytics]. (2013). Package foreach. Documentation available at CRAN:
`http://cran.r-project.org/web/packages/foreach/index.html`

Wiener, N. (1948). *Cybernetics: Control and communication in the animal and the machine*. Cambridge, MA: MIT Press.

This article was last updated on April 1, 2014.

This document was created using LaTeX